Outline

- Containers
  - About Containers
  - Container Disadvantage
  - Printing Containers
  - Container Taxonomy
Containers

• Divides the issue of "holding your objects":

1. **Collection**: a group of individual elements, often with some rule applied to them
   - **List** holds elements in a particular sequence
   - **Set** has no duplicates

2. **Map**: a group of key-value object pairs
   - Can return a **Set** of its keys
   - A **Collection** of its values
   - **Set** of its pairs
   - **Maps** can easily be expanded to multiple dimensions

All Rights Reserved.
Printing containers

- Unlike arrays, built-in functionality
- Example introduces basic container types

```java
import java.util.*;

public class PrintingContainers {
    static Collection fill(Collection c) {
        c.add("dog");
        c.add("dog");
        c.add("cat");
        return c;
    }
    static Map fill(Map m) {
        m.put("dog", "Bosco");
        m.put("dog", "Spot");
        m.put("cat", "Rags");
        return m;
    }
    public static void main(String[] args) {
        System.out.println(fill(new ArrayList()));
        System.out.println(fill(new HashSet()));
        System.out.println(fill(new HashMap()));
    }
}

>>
[dog, dog, cat]
[cat, dog]
{cat=Rags, dog=Spot}
```
Filling Containers

- `Containers.fill()`
  - duplicates a single object into the container
  - Still a problem to add interesting data
- Will use another generator
- For Maps, it needs to produce an object containing two objects: `Pair`
Container Disadvantage

• Unknown type
• Containers all hold **Object**, so they hold anything
• Collection of **Cat** will also hold a **Dog**
• Casting back to the correct type
• Throwing an exception; if you try to cast to the wrong type problem because you don’t know until the program is running
//: c09:Cat.java
public class Cat {
    private int catNumber;
    Cat(int i) { catNumber = i; }
    void print() {
        System.out.println("Cat #" + catNumber);
    }
}

//: c09:Dog.java
public class Dog {
    private int dogNumber;
    Dog(int i) { dogNumber = i; }
    void print() {
        System.out.println("Dog #" + dogNumber);
    }
}

//: c09:CatsAndDogs.java
// Simple container example.
import java.util.*;
public class CatsAndDogs {
    public static void main(String[] args) {
        ArrayList cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));
        // Not a problem to add a dog to cats:
        cats.add(new Dog(7));
        for(int i = 0; i < cats.size(); i++)
            ((Cat)cats.get(i)).print();
        // Dog is detected only at run-time
    }
}

All Rights Reserved.
Iterators

• The abstraction
  – selecting each element in a sequence (traverse a Collection)
    • Possibly make many iterators, and select more than one element at a time
    • Hides underlying collection, so you can easily change from one type to another
    • A design pattern
import java.util.*;

class Hamster {
    private int hamsterNumber;
    Hamster(int i) { hamsterNumber = i; }
    public String toString() {
        return "This is Hamster #" + hamsterNumber;
    }
}

class Printer {
    static void printAll(Iterator e) {
        while(e.hasNext())
            System.out.println(e.next());
    }
}

public class HamsterMaze {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 3; i++)
            v.add(new Hamster(i));
        Printer.printAll(v.iterator());
    }
}
Container Taxonomy
The Collection Interface

- boolean add(Object)
- boolean addAll(Collection)
- void clear()
- boolean contains(Object)
- boolean containsAll(Collection)
- boolean isEmpty()
- Iterator iterator()
- boolean remove(Object)
- boolean removeAll(Collection)
- boolean retainAll(Collection)
- int size()
- Object[] toArray()
- Object[] toArray(Object[] a)
Lists have additional abilities

- void add(index, Object)
- boolean addAll(index, Collection)
- Object get(index)
- int indexOf(Object)
- int lastIndexOf(Object)
- ListIterator listIterator()
- ListIterator listIterator(index)
- Object remove(index)
- Object set(index, Object) // Means: “replace”
- List subList(fromIndex, toIndex)
Lists produce ListIterators

- boolean hasNext()
- Object next()
- boolean hasPrevious()
- Object previous()
- int nextIndex()
- int previousIndex()
- add(Object)
- void remove()
- void set(Object) // “Replace”
ArrayList & LinkedList

- **ArrayList**
  - Default choice for a simple sequence
  - Optimal for rapid random access
  - `ListIterator` should be used only for back-and-forth traversal of an *ArrayList*, but not for inserting and removing elements

- **LinkedList**
  - Optimal sequential access with inexpensive insertions and deletions from the middle of the list
  - possibly be used as a stack, a queue, and a deque
  - `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, and `removeLast()`
  - `ListIterator` can be used for insertion and removal
• Not there
• Use **LinkedList**
  – `addFirst( )`, `addLast( )`
  – `removeFirst( )`, `removeLast( )`
  – `getFirst( )`, `getLast( )`
Sets

- Mathematical set abstraction
- No duplicates
- **Two interfaces**
  - **Set** (identical to Collection)
  - **SortedSet**
- **Implementations**
  - **HashSet** for maximally fast lookups
  - **TreeSet** to maintain sorted order
    - Order determined by **Comparable** or **Comparator**
    - You can extract an ordered sequence from a **TreeSet**:
      
      ```java
      Comparator comparator()
      Object first()
      Object last()
      SortedSet subSet(fromElement, toElement)
      SortedSet headSet(toElement)
      SortedSet tailSet(fromElement)
      ```
import java.util.*;
import com.bruceeckel.util.*;
public class Set1 {
    static Collections2.StringGenerator gen = Collections2.countries;
    public static void testVisual(Set a) {
        Collections2.fill(a, gen.reset(), 10);
        Collections2.fill(a, gen.reset(), 10);
        Collections2.fill(a, gen.reset(), 10);
        System.out.println(a); // No duplicates!
        // Add another set to this one:
        a.addAll(a);
        a.add("one");
        a.add("one");
        a.add("one");
        System.out.println(a);
        // Look something up:
        System.out.println("a.contains("one")": " + a.contains("one");
    }
    public static void main(String[] args) {
        System.out.println("HashSet");
        testVisual(new HashSet());
        System.out.println("TreeSet");
        testVisual(new TreeSet());
    }
}
Maps

- Alternately termed a *map*, a *dictionary* or an *associative array*
- Like an *ArrayList*, but instead of using an integral index to look something up, another object is used as a *key*
- Store key/value pairs of *Objects*
- Duplicate Keys not allowed
- Returns keys as a *Set* view
- Returns values as a *Collection*
TreeMap

- **SortedMap** with elements stored in a tree
  - Comparator comparator()
  - Object firstKey()
  - Object lastKey()
  - SortedMap subMap(fromKey, toKey)
  - SortedMap headMap(toKey)
  - SortedMap tailMap(fromKey)

- Viewing keys or pairs will be in sorted order
- Order determined by **Comparable** or **Comparator**
HashMap

• HashMap
  – a Map with very fast lookup: uses a hash code to organize the objects
• Should typically be your first choice for a Map
• Program counts occurrences of “random” numbers:
import java.util.*;

class Counter {
    int i = 1;
    public String toString() {
        return Integer.toString(i);
    }
}

public class Statistics {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        for(int i = 0; i < 10000; i++) {
            // Produce a number between 0 and 20:
            Integer r =
                new Integer((int)(Math.random() * 20));
            if(hm.containsKey(r))
                ((Counter)hm.get(r)).i++;
            else
                hm.put(r, new Counter());
        }
        System.out.println(hm);
    }
}
HashMap “key” classes

• A common pitfall
  – if a class is to be used as a key for a HashMap, you must override `hashCode( )` and `equals( )` from the common root class `Object`
Old 1.0/1.1 Containers

- They’ll appear in older standard libraries
  - And sometimes new ones
- Don’t use them in new code
  - Vector -> ArrayList
  - BitSet (maybe use)
  - Stack -> LinkedList
  - Hashtable -> Map
Enumeration: old Iterator

- Smaller interface than `Iterator`, longer method names
  - `nextElement()` moves to and produces the next item
  - `hasMoreElements()` indicates the end
- Can actually appear quite a bit, even in new library code
- Treat it just like an `Iterator` without a `remove()` method
Summary

- **Sequence (List)** associates numerical indices to objects
  - Only holds **Object** references
  - Automatically resizes itself
- **Set** only holds one of each object
  - **HashSet** for speed, **TreeSet** for sorted order
- **Map** associates **objects** with other objects
  - Provides rapid random access
  - **HashMap** orders with **hashCode()** & **equals()**
  - **TreeMap** keeps sorted order